Intro to Recurrent Networks

AW

Lecture Overview

1. Time Series Data Sets
2. DDS and Computational Graphs
3. RNN Architectures
4. Teacher Forcing

RNN

Recurrent Neural Networks operate on sequential data. Typical task to predict next value in a sequence:

- Record of stock performance (prices) over time. Goal predict ‘next’ price (after sequence ends)
- Text (sequence of words). Goal predict next word
- DNA sequence of amino acids. Predict next element in the sequence

All examples are *time series*, i.e. series of data points indexed (or listed or graphed) by natural numbers. Index of a data point is its ‘time-stamp’

- In text it is the word number in the sequence of words
- In DNA it is position number of amino acid
- In stock performance number it is the first record of the observation

Problems for Neural Networks

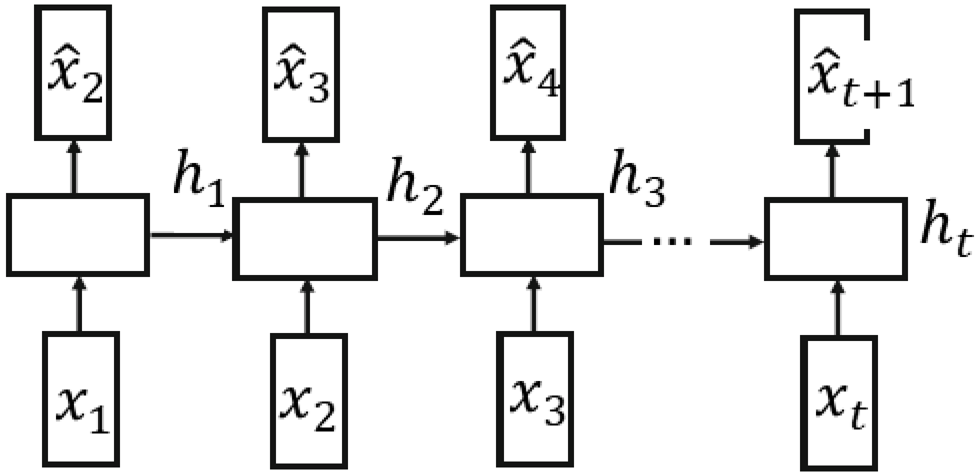
- Neural network can only process fixed size (length) inputs. Sequences have variable size
- Next element of a sequence is not drawn independently from a set of possible instances. Instead it depends on previous elements of the sequence and their order. Neural nets so far had no memory

Same problems as in statistics. Methods to address:

- Autoregressive models. Limit modeling dependencies to fixed number τ of previous stages, i.e. if the sequence is $x_1, x_2, \dots, x_t$ to predict x_{t+1} we do not need the whole sequence – we only need subsequence $x_{t-\tau+1}, \dots, x_t$ of fixed length
- Latent Autoregressive models. Recording some summary h_t of the past observations, and at the same time update summary h_t to obtain h_{t+1} in addition to the prediction $\hat{x}_{t+1}$.

Possible Architecture For Sequences

- If we have sequence is $x_1 \; x_2 \ldots x_t$ then using latent autoregressive idea the architecture should be

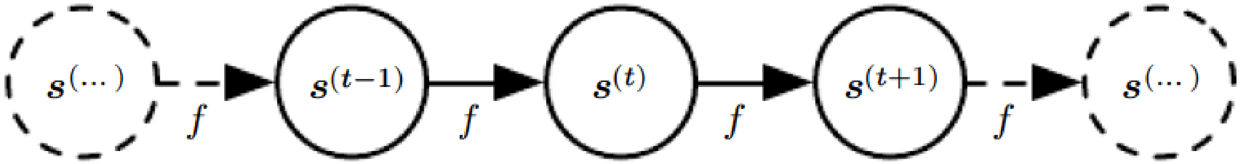


- It is still dependent on the length of the sequence. Need to fold rightward expansion How can this be avoided?
- The idea is to share weights across the architecture. How?
- Take clues of Discrete Dynamical Systems (DDS)

Lecture Overview

DDS and its Unfolding

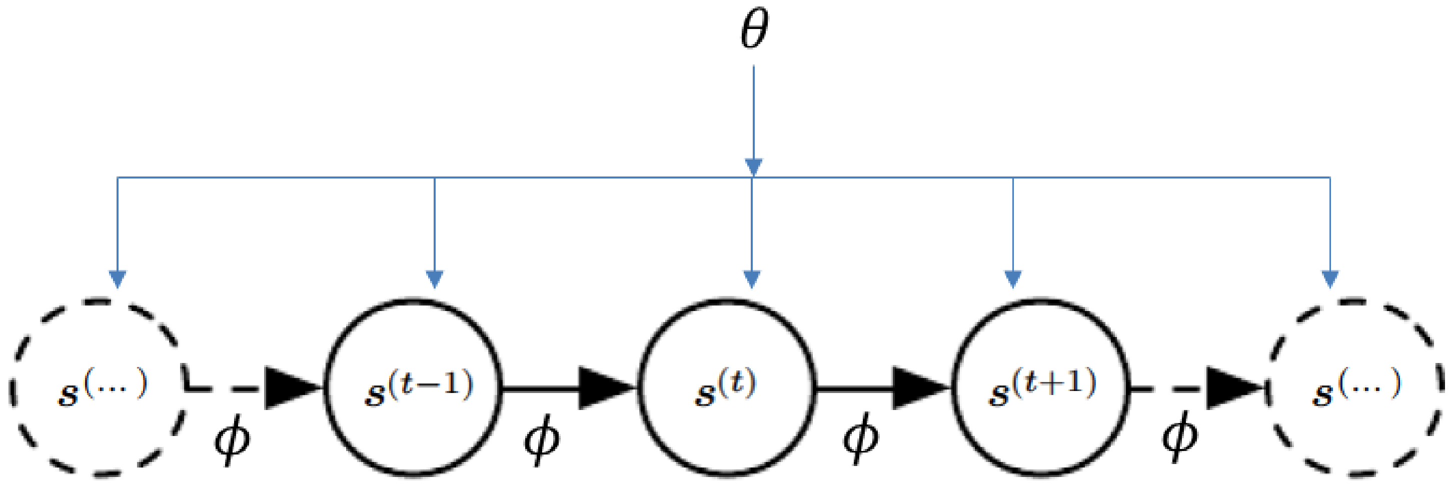
- Let T be the set of integers, M a ground set aka *state space*, and Φ is a map called *evolution function* that maps $T \times M \to M$ such that $\Phi(0, s) = s$, and $\Phi(t_1 + t_2, s) = \Phi(t_2, \Phi(t_1, s))$ for all $s \in M$
- We write s_0 for $\Phi(0, s)$ and s_t for $\Phi(t, s)$ then $s_{t_1+t_2} = \Phi(t_2, s_{t_1})$ and in particular $s_t = \Phi(1, s_{t-1}) = f(s_{t-1})$ where $f(s) = \Phi(1, s)$ for all s. How would computational graph of the systems look like?



- What if we are given a family of dynamical systems parameterized by some parameter space Θ. Then once we choose the parameter value $\theta \in \Theta$ we get dynamical system $s_t = f_\theta(s_{t-1}) = \phi(s_{t-1}, \theta)$

DDS and Succinct Computational Graphs

- How does computation of $s_t = \phi(s_{t-1}, \theta)$ look as a graph?

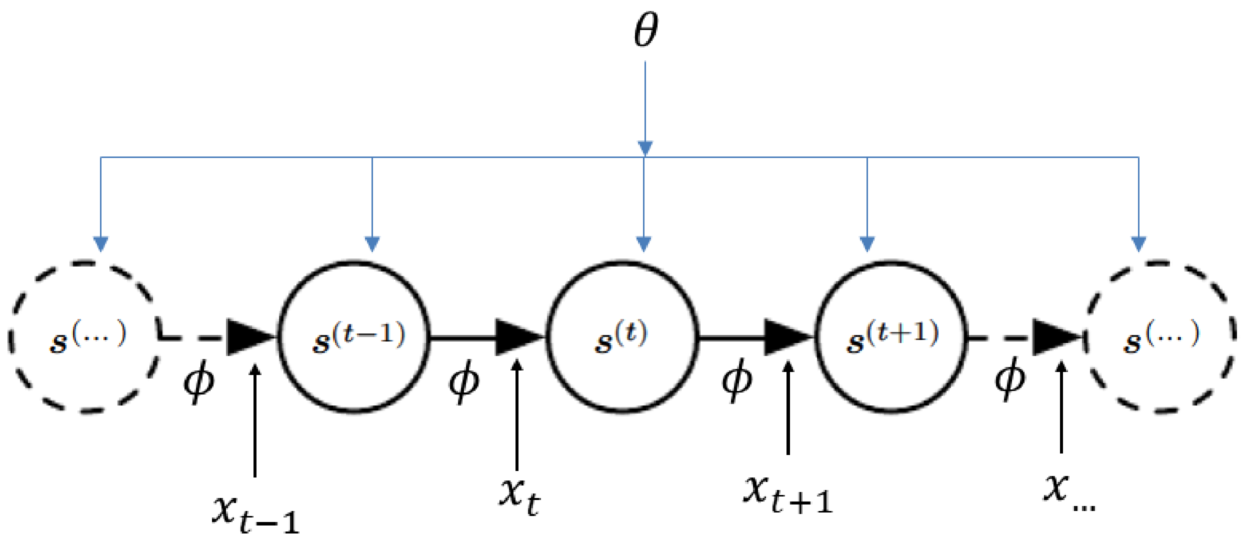


- Note that single parameter θ is shared between all computation
- What if in our dynamical systems some external perturbation x_t that changes with time is added at every step and the resulting state depends on the added perturbation?

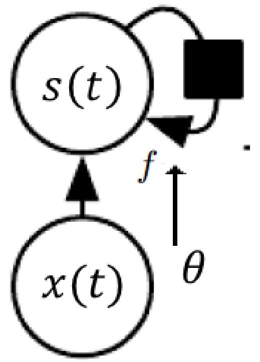
  Then the DDS looks like $s_t = \phi(s_{t-1}, x_t, \theta)$

DDS and Succinct Computational Graphs (cont.)

Function $s_t = \phi(s_{t-1}, x_t, \theta)$ has computational graph

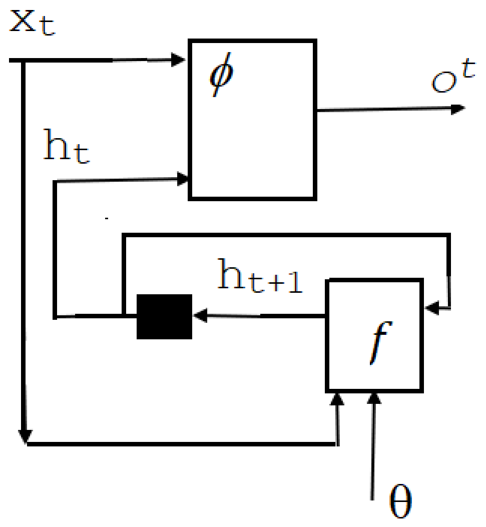


Let's introduce a new element into a computational graph: time delay unit depicted by a black box. The above graph can be expressed as (folded) into succinct graph

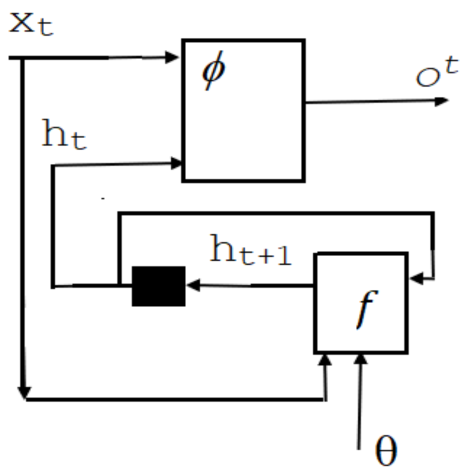


Recurrent Neural Nets – Succinct Graphs

- We need to compute $o^t(\vec{x}_1, \ldots, \vec{x}_t)$ for every t where arguments are taken from the input sequence of vectors $\vec{x}_1, \ldots, \vec{x}_t, \ldots.$
- So these $\vec{x}_1, \ldots, \vec{x}_t, \ldots.$ are going to be the input nodes
- We'd like to use the idea of latent autoregression, so for hidden nodes we want to compute history vectors $\vec{h}^t = g^t(\vec{x}_1, \ldots, \vec{x}_t)$ such $o^t = \phi(\vec{h}^t, x_t)$ and we want $g^t(\vec{x}_1, \ldots, \vec{x}_t)$ be such that it can be computed as $h^t = f(\vec{h}^{t-1}, \vec{x}_t, \vec{\theta})$ for some parameter $\vec{\theta}$.



Recurrent Neural Nets – Succinct Graphs



- But that is a DDS! So we can fold it, which has major advantages:
  - Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state vector to another, rather than in terms of a variable-length history of inputs
  - It is possible to use the *same* transition function *f* with the same parameters at every time step

Recurrent Neural Nets – Weights

Since these are neural networks parameters are weights and biases so these functions f and ϕ must be defined in terms of weights, biases and activation functions. So,

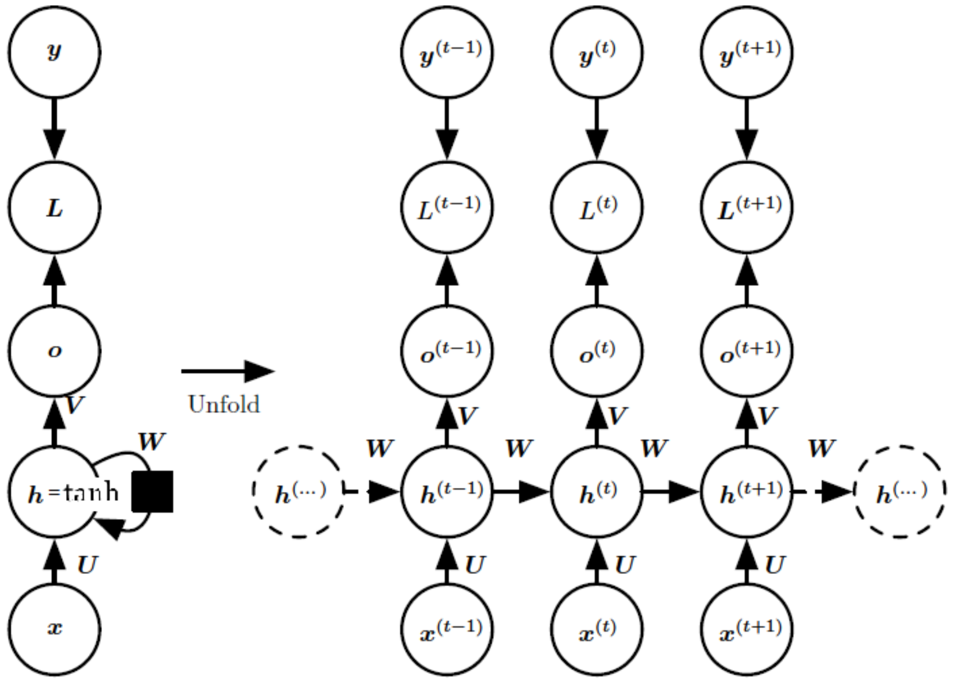
- For computing f since all $x, h,$ are in fact vectors we should have:
  - Matrix of weights W for states
  - Matrix of weights U for inputs
  - Bias b for computing linear input of f
  - Acrtivation function for f
- For computing ϕ we must have
  - Matrix of weights V for states
  - Matrix of weights M for inputs. However we might define output only in terms of state space since states already incorporate inputs. If so M is not needed
  - Biases c for outputs

Lecture Overview

1. Time Series Data Sets
2. DDS and Computational Graphs
3. RNN architectures
4. Teacher Forcing

Example of RNN Architecture

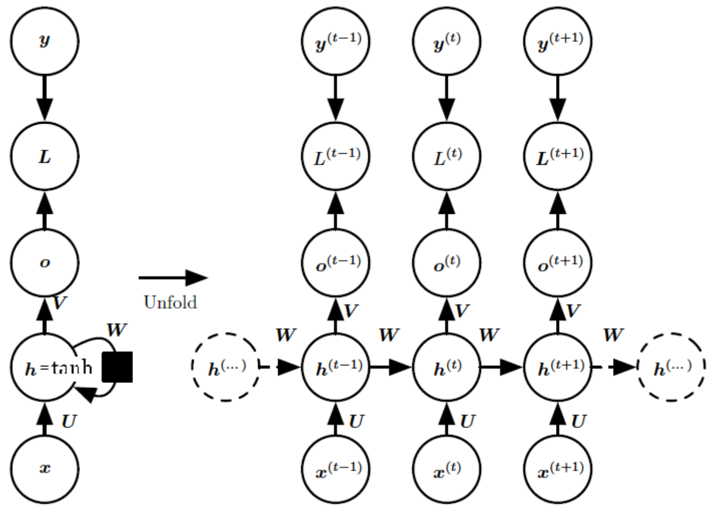
- RNN with output at each step that have recurrent connections between hidden units with computational graphs:



- Left succinct graph, right unfolded graph
- where $\vec{y}$ is target variable, L is loss function, $\vec{o}$ is output, $\vec{h}$ is hidden unit value, W is hidden-to-hidden weight matrix, U is input-to-hidden weight matrix, V is hidden-to output weight matrix, $\vec{x}$ is input sequence
- W, V and U are RNN parameters

Example of RNN Architecture

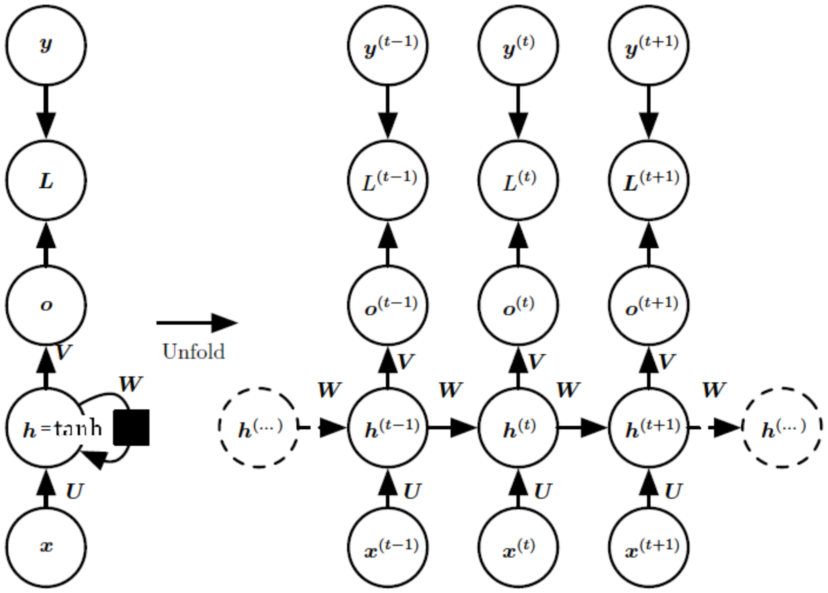
- RNN with output at each step that have recurrent connections between hidden units with computational graphs:



The RNN with this architecture (and recurrence relations that we'll see shortly) is universal – it can simulate universal TM. So

- any function computable by a Turing machine can be computed by such a recurrent network of a finite size.
- The output can be read from the RNN after a number of time steps that is asymptotically linear in the number of time steps of the Turing machine and is asymptotically linear in the length of the input

Example of RNN Based on this Architecture

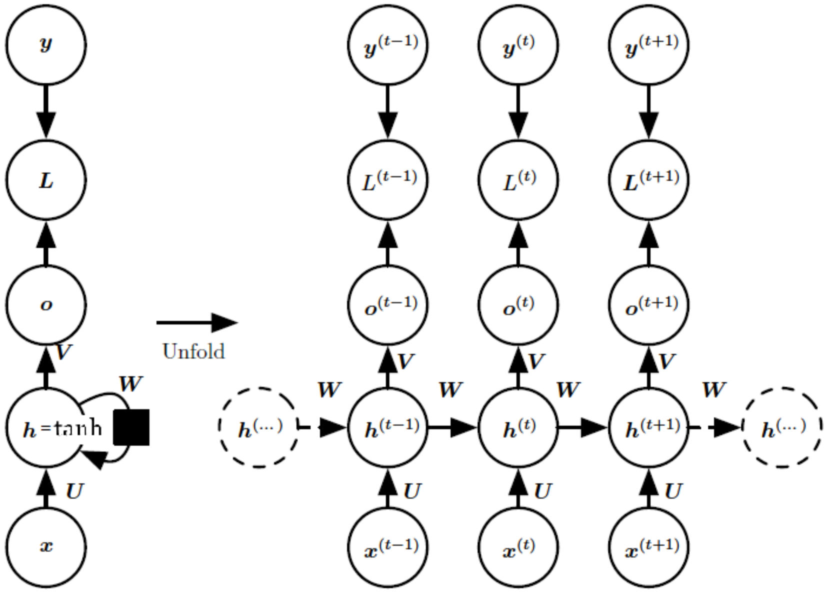


This figure is not RNN since it does not specify what x, o, L, y are, so make it RNN assume

Example:

- $\mathrm{x} =< \vec{x}_1, \dots, \vec{x}_t, \dots >$ is a text with each word represented by a vector (will talk about word2vec later)
- o is unnormalized vector of $\log$ probability distribution over possible values of the next word vector
- L is cross-entropy assuming decision is by softmax using $\hat{y}$ predicted value vector of normalized probabilities computed from o

Example of RNN: Forward Propagation

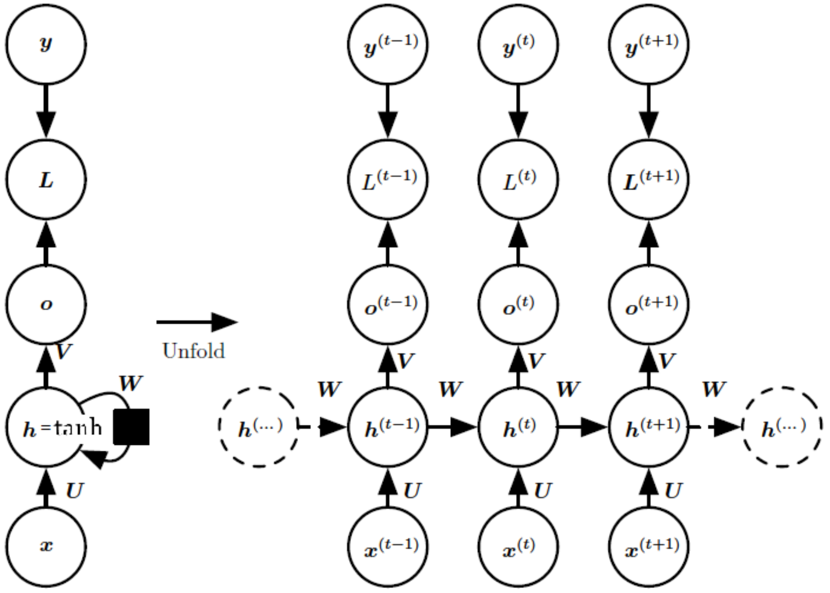


- Initial condition specification: $\vec{h}^{(0)}$
- Recurrence relations:

$$\begin{aligned} a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)} \\ h^{(t)} &= \tanh(a^{(t)}) \\ o^{(t)} &= c + V\,h^{(t)} \\ \hat{y}^{(t)} &= \mathrm{softmax}(o^{(t)}) \end{aligned}$$

- where the parameters are the bias vectors $\vec{b}$ and $\vec{c}$ and the weight matrices U , V and W

Example of RNN: Forward Propagation (cont.)

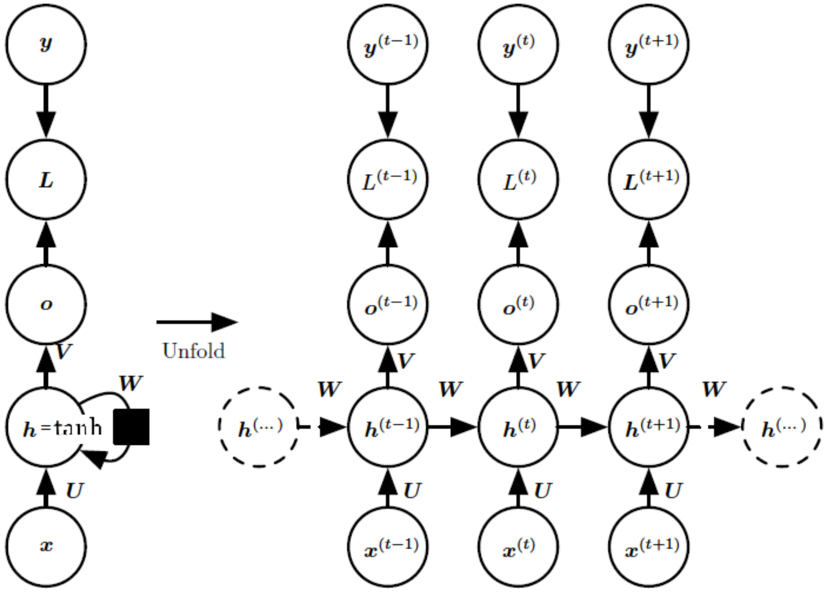


RNN: $\vec{h}^{(0)}$;

$$\begin{aligned} a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)}; \\ h^{(t)} &= \tanh(a^{(t)}); \\ o^{(t)} &= c + V\,h^{(t)} \\ \hat{y}^{(t)} &= \text{softmax}(o^{(t)}) \end{aligned}$$

- The total loss for a given pair of sequences $(\mathrm{x}, \mathrm{y}) = (< \vec{x}_1, \dots, \vec{x}_t, \dots, \vec{x}_\tau >, < y_1, \dots, y_t, \dots, y_\tau >)$ is then just the sum of the losses over all time steps. Since individual loss is cross-entropy $L(\mathrm{x}, \mathrm{y}) = \sum_{t=1}^{\tau} L(t) = -\sum_{t=1}^{\tau} \log p(y_t | x_1, \dots, x_t)$

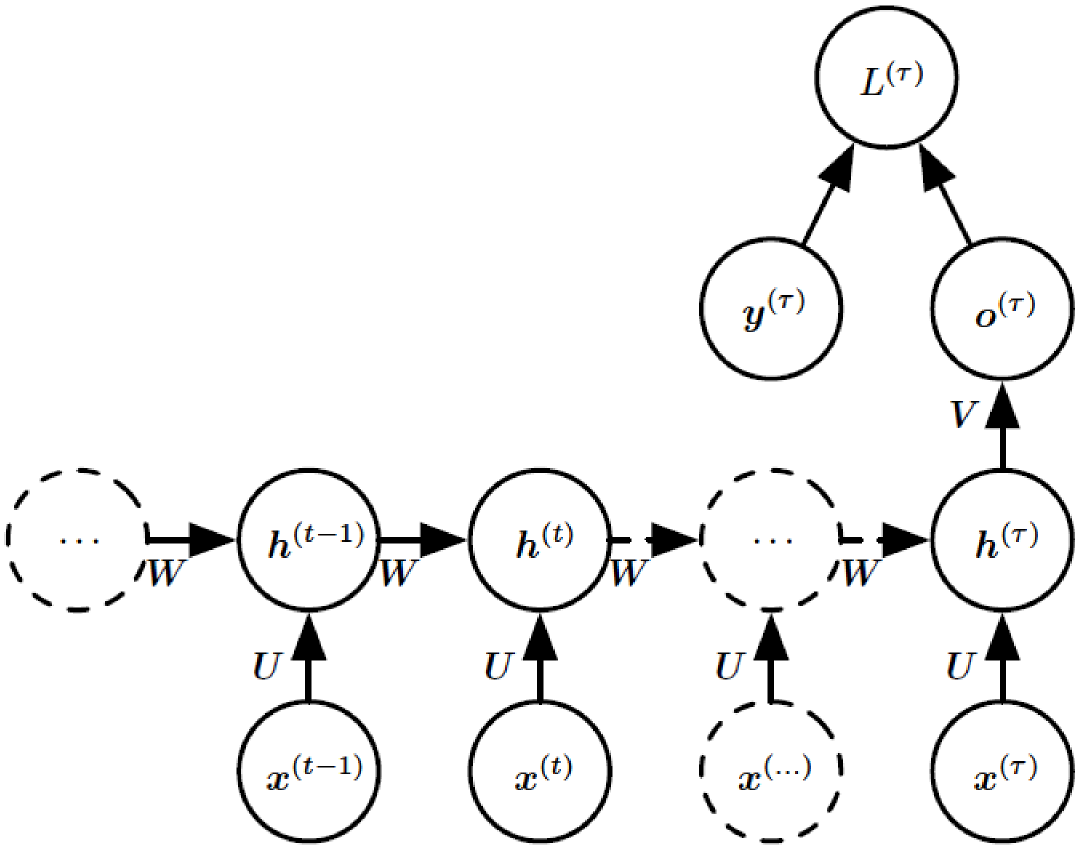
Example of RNN: Forward Propagation (cont.)



- For an input sequences of length τ runtime of forward propagation is $O(\tau)$.
- It cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may only be computed after the previous one.
- States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also $O(\tau)$.
- The back-propagation algorithm is called *back-propagation through time(BPTT)* (later) and is computed by applying to unrolled graph
- *Computationally expensive!*

Another Example of RNN Architecture

Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output.

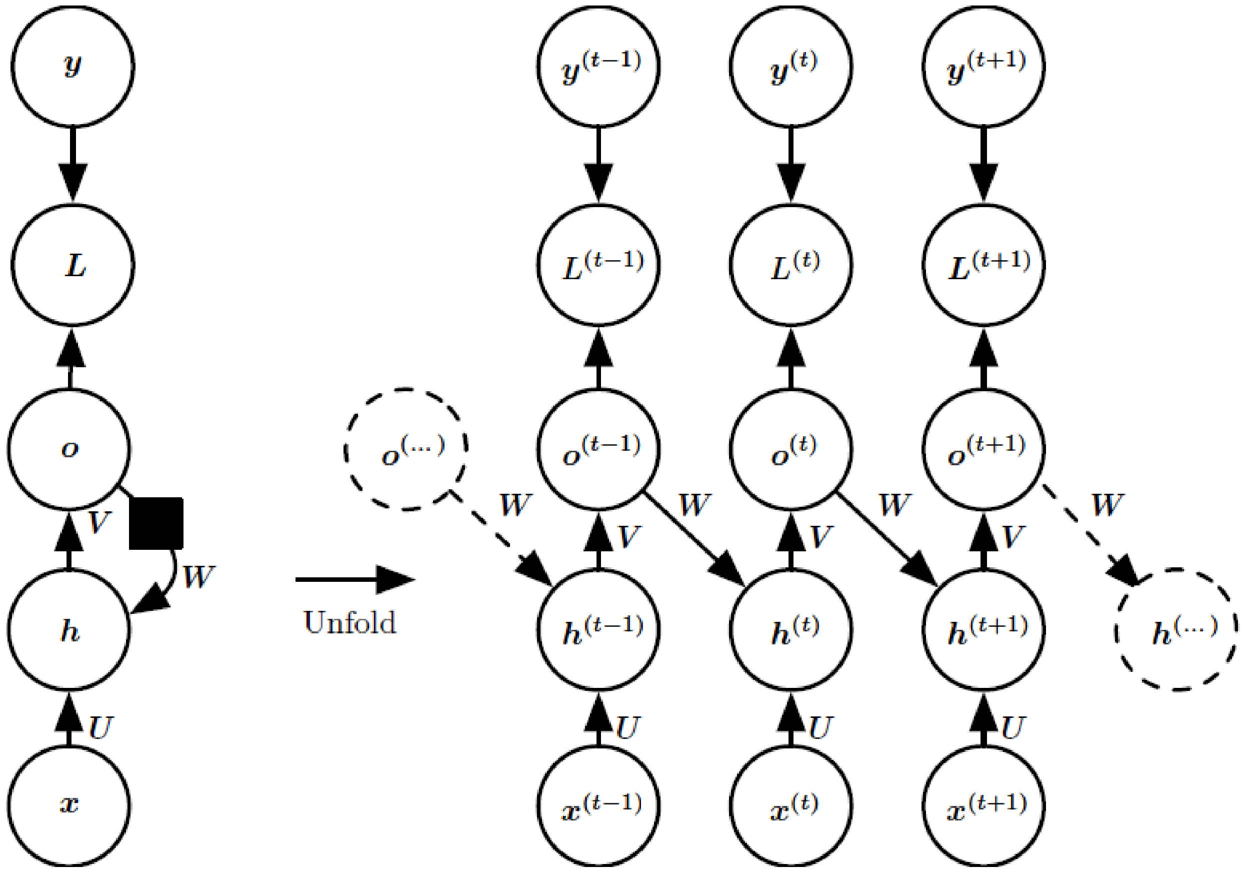


Lecture Overview

1. Time Series Data Sets
2. DDS and Computational Graphs
3. RNN architectures
4. **Teacher Forcing**

Teacher Forcing: Another RNN Architecture

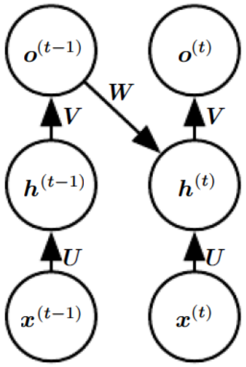
RNN that produces output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step. It is strictly less powerful than universal TM (because of lack of hidden-to hidden connections)



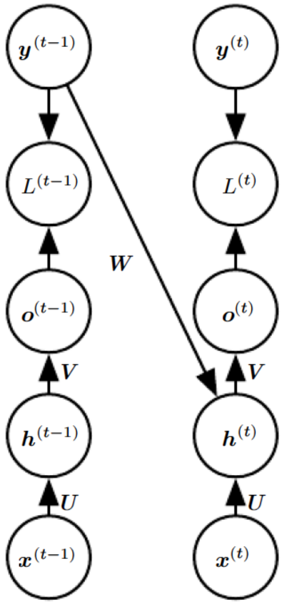
Teacher Forcing is Time Decoupled!

No hidden-to-hidden recurrence ⇒ for a loss function based on comparing the prediction at time t to the training target at time t there is no need to know output of previous step to compute output at current because the training set has best possible value for output of previous step.

Instead of



at train time do



- Training parallelized, with the gradient for each step t computed in isolation.

Teacher Forcing as a Procedure

- *Teacher Forcing* is a procedure in which during training the model receives the ground truth output $y^{(t)}$.
- It is the ideal procedure when we are looking for cross entropy. Consider 2 step sequence training: the model is trained to maximize the conditional probability of $y^{(2)}$ given *both* the $\mathrm{x} = \langle x(1), x(2)\rangle$ sequence so far and the previous $y^{(1)}$ value:

$$\log p\big(\langle y^{(1)}, y^{(2)}\rangle \mid \langle x^{(1)}, x^{(2)}\rangle\big)$$
$$= \log p(y(2) \mid y\ (1), \langle x(1), x(2)\rangle) + \log p(y(1) \mid \langle x(1), x(2)\rangle)$$

But of course it is maximized when it is taken from the training set rather than from model output.

- Teacher forcing can be applied to models that have hidden-to-hidden connections too, as long as they have connections from the output at one time step to values computed in the next time step.
- When hidden units become a function of earlier time steps, teacher forcing is not applicable - the BPTT algorithm is necessary.

Reading

- Ch. 7.1
- GBC, Ch. 10.1, 10.2.1